Application du Pattern Factory

Ce TP reprend et étend la réalisation de l'application faite dans le TP N°1 sur la gestion du trafic maritime.

Exercice 1 :

Les bateaux vont être désormais équipés de moteur et induire ainsi une vitesse (i.e le nombre de cases dans le tableau) à laquelle pourront se déplacer les instances des classes Ship.

1°) Ecrire une classe **Engine** comportant une méthode *getSpeed()* **virtuelle pure**.

2°) Faire Hériter la classe **Engine** par 2 autres classes moteurs de votre choix. Ré-implémenter la méthode *getSpeed()*.

Les objets vont être également équipés d'une coque. Les coques sont définies par la qualité de leur solidité pour prévenir des éventuels chocs lors d'une mauvaise navigation dans la matrice.

3°) Ecrire une classe **Hull** définie par une solidité (*m_solidity)*. La classe comportera les méthodes suivantes :

- *int getSolidity()* qui retourna la valeur de solidité.
- *bool operator < (const Hull& a)* **virtuelle pure** prenant en entrée une référence constante sur un autre objet de type **Hull**.

4°) Faire Hériter la classe **Hull** par 2 autres classes de votre choix. Ré-implémenter la méthode *bool operator < (const Hull& a).*

5°) Créer une Factory abstraite proposant des méthodes virtuelles pures pour créer des objets de type **Hull** et de type **Engine.**

6°) Faire hériter votre Factory par 2 autres classes Factory. Ré-implémenter les méthodes de création sachant que l'ensemble des classes créées précédemment dans les questions *2°)* et *4°)* doivent faire l'objet d'au moins une création instance dans l'une des implémentations de la Factory.

Exercice 2 :

1°) Modifier la classe **Ship** afin de faire apparaître dans sa déclaration une classe **Hull** et une classe **Engine**. Créer dans la classe **Ship** la méthode *getSpeed()* afin d'appeler celle de la classe **Engine.**

2°) *Les objets de type Ship sont donc susceptibles d'entrer en collision lors des déplacements.*

Dans la classe **Harbor,** écrire une méthode *collision(Ship*s1, Ship* s2)* qui devra comparer les instances de la classe **Hull** de chacun des 2 objets entrant en collision. Cette méthode

devra être appelée à chaque fois qu'un bateau est déplacé sur une case déjà occupée par une autre instance.

N.B : A l'issu de la comparaison, si l'objet percuté est le plus faible, il sera détruit et l'espace libéré. Sinon, l'objet percutant devra trouver un itinéraire bis…

3°) Modifier la classe **Ship** en ajoutant les méthodes virtuelles pures suivantes :

- *float failureProbability() = 0 ; cette méthode détermine la probabilité d'un bateau de s'arrêter subitement lors d'un cycle de déplacement.*
- *bool accept(int dockId) = 0 ; cette méthode indique si oui ou non le bateau accepte le numéro du quai qui lui a été affecté.*
- *int getPriority() =0 ; cette méthode donne une priorité différente en fonction du type de bateau.*

4°) Créer 4 classes différentes qui héritent de la classe **Ship.** L'implémentation des méthodes précédentes devra respecter les consignes données dans tableau ci-dessous**.**

Nom de la classe	*failureProbability*	*accept(dockId)*	*priority*
PassengerShip	20%	$0 \leq$ dockId ≤ 10	7
MilitaryShip	50%	si dockId > 20	10
PleasureCraft	5%	Tous les docks Id	2
FishingBoat	Aléatoire entre 9 et 99%	Les dockId impairs	4

Les classes devront prendre en paramètre du constructeur un pointeur sur la Factory créée dans l'exercice précédent. Les objets **Engine** et **Hull** de chacune des classes seront instanciés à partir de la Factory passée en paramètre.

Exercice 3 :

1°) Transformer la méthode de génération d'objet **Ship** (cf. Exercice 3 du TP1) en une méthode de factory qui permet de créer un type de vaisseau différent de manière aléatoire.

2°) Modifier la méthode *cycle* afin qu'elle adopte ce nouveau comportement:

- avant d'affecter un numéro de quai à un vaisseau, elle appelle la méthode *accept* du l'objet *Ship* en cours. Tant que l'appel échoue, elle renouvelle sa demande avec un autre numéro de quai. Si tous les quais « acceptables » sont pris, alors l'objet *Ship* ayant une priorité inférieur à l'objet arrivant dans la matrice devra laisser sa place. (suppression de l'instance en mémoire et libération de la place).
- Prendre en compte, lors du déplacement de chaque objet, leur probabilité de rencontrer une avarie (*failureProbability)* : si l'issu est réalisée, l'objet garde sa position jusqu'au prochain cycle.
- Prendre en compte lors de chaque déplacement la valeur de la vitesse correspondant au nombre de cases dont l'objet peut se déplacer en 1 cycle.